Performance analysis of SCOOP programs

Benjamin Morandi[a,*], Sebastian Nanz[a], Bertrand Meyer[a]

[a] *Chair of Software Engineering, ETH Zurich, Clausiusstrasse 59, 8092 Zurich, Switzerland*

Abstract

To support developers in writing reliable and efficient concurrent programs, novel concurrent programming abstractions have been proposed in recent years. Programming with such abstractions requires new analysis tools because the execution semantics often differs considerably from established models. We present a performance analyzer that is based on new metrics for programs written in SCOOP, an object-oriented programming model for concurrency. We discuss how the metrics can be used to discover performance issues, and we use the tool to optimize a concurrent robotic control software.

Keywords: performance analysis, performance metric, profiling, tracing, concurrent programming, SCOOP

1. Introduction

Avoiding concurrency-specific errors such as data races and deadlocks is still the responsibility of developers in most languages that provide synchronization through concurrency libraries. To avoid the problems of the library approach, a number of languages have been proposed that integrate synchronization mechanisms. SCOOP (Simple Concurrent Object-Oriented Programming) (Meyer, 1997; Nienaltowski, 2007), an object-oriented programming model for concurrency, is one of them.

The main idea of SCOOP is to simplify the writing of correct concurrent programs, by allowing developers to use familiar concepts from object-oriented programming, but protecting them from common concurrency errors such as data races. Empirical evidence supports the claim that SCOOP indeed simplifies reasoning about concurrent programs as opposed to more established models (Nanz et al., 2011). The advantages of the model are due to a runtime system that automatically takes care of operations such as obtaining and releasing locks, without the need for explicit program statements.

The complex interactions between concurrent components make it difficult to analyze the behavior of concurrent programs. Effective use of a programming model therefore requires tools to help developers analyze and improve programs. Static analysis of models, e.g., Ostroff et al. (2008); Brooke et al. (2007); West et al. (2010); Nanz et al. (2008), can establish some degree of functional correctness. However, they fail to explain why a particular execution is slow, and they do not help choosing optimal execution parameters. Addressing such issues requires adapting performance analysis techniques to the context of concurrent, non-deterministic execution. Section 6 surveys existing tools that address this goal in the context of threading and various other concurrency models. They are not appropriate, however, for the semantics of SCOOP, which requires different approaches for measuring and visualization. For example, SCOOP programs go through synchronization steps to lock resources and establish conditions on these resources; a performance analyzer for SCOOP must take this into account.

We present a performance analyzer for SCOOP programs. The main contributions are performance metrics for SCOOP and a technique to compute them from event traces. The resulting tool has been integrated into the EVE development environment (ETH Zurich, 2012a), which we extended with support for SCOOP; it can be downloaded from the SCOOP website (ETH Zurich, 2012b). We evaluate the metrics and the tool on a number of example problems, as well as on a larger case study on optimizing a robotics control software written in SCOOP. To the best of our knowledge, this work is the first to suggest performance metrics for SCOOP.

This article is structured as follows. Section 2 gives an overview of the SCOOP model. Section 3 introduces the metrics and shows how to calculate them from events. Section 4 describes the tool built around the metrics. Section 5 analyzes the time overhead of the tool and shows how to optimize a concurrent robotic control software using the tool. Finally, Section 6 provides an overview of related work and Section 7 concludes with an outlook on future work.

2. SCOOP

This section gives an overview of SCOOP.

2.1. Introduction to SCOOP

The starting idea of SCOOP is that every object is associated for its lifetime with a processor, called its *handler*. A *processor* is an autonomous thread of control capable of executing actions on objects. An object's class describes the possible actions as *features*. A processor can be a CPU, but it can also be implemented in software, for example as a process or as a thread; any

*Corresponding author. Tel.: +41-44-632-7828; fax: +41-44-632-1435

Email addresses: `benjamin.morandi@inf.ethz.ch` (Benjamin Morandi), `sebastian.nanz@inf.ethz.ch` (Sebastian Nanz), `bertrand.meyer@inf.ethz.ch` (Bertrand Meyer)

mechanism that can execute instructions sequentially is suitable as a processor.

A variable *x* belonging to a processor can point to an object with the same handler (*non-separate object*), or to an object on another processor (*separate object*). In the first case, a *feature call x.f* is *non-separate*: the handler of *x* executes the feature synchronously. In this context, *x* is called the *target* of the feature call. In the second case, the feature call is *separate*: the handler of *x*, i.e., the *supplier*, executes the call asynchronously on behalf of the requester, i.e., the *client*. The possibility of asynchronous calls is the main source of concurrent execution. The asynchronous nature of separate feature calls implies a distinction between a feature call and a *feature application*: the client logs the call with the supplier (feature call) and moves on; only at some later time will the supplier actually execute the body (feature application).

The producer-consumer problem serves as a simple illustration of these ideas. A root class defines the entities *producer*, *consumer*, and *buffer*. Assume that each object is handled by its own processor. One can then simplify the discussion using a single name to refer both to the object and its handler. For example, one can use “producer” to refer both to the producer object and its handler.

```
producer: separate PRODUCER
consumer: separate CONSUMER
buffer: separate BUFFER [INTEGER]
```

The keyword **separate** specifies that the referenced objects may be handled by a processor different from the current one. A *creation instruction* on a separate entity such as *producer* will create an object on another processor; by default the instruction also creates that processor.

Both the producer and the consumer access an unbounded buffer in feature calls such as *buffer.put* (*n*) and *buffer.item*. To ensure exclusive access, the consumer must lock the buffer before accessing it. Such locking requirements of a feature must be expressed in the formal argument list: any target of separate type within the feature must occur as a formal argument; the arguments' handlers are locked for the duration of the feature execution, thus preventing data races. Such targets are called *controlled*. For instance, in *consume*, *buffer* is a formal argument; the consumer has exclusive access to the buffer while executing *consume*.

Condition synchronization relies on preconditions (after the **require** keyword) to express wait conditions. Any precondition makes the execution of the feature wait until the condition is true. For example, the precondition of *consume* delays the execution until the buffer is not empty. As the buffer is unbounded, the corresponding producer feature does not need a precondition.

```
consume (buffer: separate BUFFER [INTEGER])
    -- Consume an item from the buffer.
  require
    not (buffer.count = 0)
  local
    consumed_item: INTEGER
  do
    consumed_item := buffer.item
  end
```

During a feature call, the consumer could pass its locks to the buffer if it has a lock that the buffer requires. This mechanism is known as *lock passing*. In such a case, the consumer would have to wait for the passed locks to return. In *buffer.item*, the buffer does not require any locks from the consumer; hence, the consumer does not have to wait due to lock passing. However, the runtime system ensures that the result of the call *buffer.item* is properly assigned to the entity *consumed_item* using a mechanism called *wait by necessity*: while the consumer usually does not have to wait for an asynchronous call to finish, it will do so if it needs the result.

2.2. SCOOP runtime

The SCOOP concepts require execution-time support, known as the SCOOP runtime. The following description is abstract; actual implementations may differ.

Each processor maintains a *request queue* of requests resulting from feature calls on other processors. A non-separate feature call can be processed right away without going through the request queue: the processor creates a *non-separate feature request* and processes it right away using its call stack. When the client executes a separate feature call, it enqueues a *separate feature request* to the supplier's request queue. The supplier will process the feature requests in the order of queuing.

Special attention is required in the case of *separate callbacks*, which occur for example if the buffer performs a separate feature call on the consumer, which already has a lock on the buffer. Enqueuing a feature request on the consumer could cause a deadlock if the separate callback is synchronous since the consumer might already be waiting for the buffer. Figure 1 illustrates this issue. The solution is to add such feature requests, corresponding to separate callbacks, ahead of all others in the request queue. This ensures that the consumer can process the feature request right away and the buffer can continue.

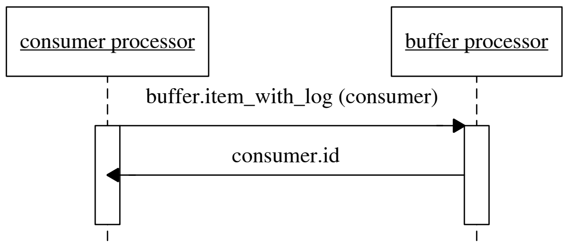


Fig. 1: A deadlock scenario based on incorrect handling of separate callbacks

The runtime system includes a *scheduler*, which serves as an arbiter between processors. When a processor is ready to process a feature request in its request queue, it will only be able to proceed after the request is satisfiable. In a *synchronization step*, the processor tries to obtain the locks on the arguments' handlers in a way that the precondition holds. For this purpose, the processor sends a *locking request* to the scheduler, which stores the request in a queue and schedules satisfiable requests for application. Once the scheduler satisfies the request, the processor starts an *execution step*.

Whenever the processor is ready to let go of the obtained locks, i.e., at the end of its current feature application, it issues an unlock request to each locked processor. Each locked processor will unlock itself as soon as it processed all previous feature requests. In the example, the producer issues an unlock request to the buffer after it issued a feature request for *put*.

The scheduler used for this work is a dedicated thread of control (Nienaltowski, 2007). It guarantees that a satisfiable locking request gets approved before any other locking request that arrives later. To ensure this, the scheduler iterates through its queue until it finds a satisfiable locking request, in which case it approves that request, removes the request, and continues. Once a processor unlocks, the scheduler restarts from the beginning of the queue to give a chance to earlier locking requests that now have become satisfiable. While this scheduler ensures a basic level of fairness, its performance is suboptimal. The scheduler approves two locking requests with disjoint sets of locks one after the other, although it could approve them in parallel. Furthermore, the scheduler is central and can thus become a bottleneck. These runtime issues have an impact on the performance of a program. To overcome some of the issues, Eiffel Software (2012) recently proposed a decentralized scheduler, to which we are currently applying this work.

3. Metrics

This section first defines the metrics for SCOOP along with a discussion on how the metrics can be used to diagnose issues in concurrent programs and find appropriate solutions. It then describes a way to calculate the metric values.

3.1. Definitions

To define SCOOP-specific metrics, one must take a closer look at the interactions in the runtime system. Consider a share market application with investors, share issuers, and markets, where integer identifiers represent the issuers. The following listing shows the class that describes the investors. The market and the investors are handled by different processors.

```
class INVESTOR feature
  id: INTEGER

  buy (market: separate MARKET; issuer_id: INTEGER)
      −− Buy a share of the issuer on the market.
    require
      market.can_buy (id, issuer_id)
    do
      market.buy (Current, issuer_id)
    end

  sell (market: separate MARKET; issuer_id: INTEGER)
      −− Sell a share of the issuer on the market.
    require
      market.can_sell (id, issuer_id)
    do
      market.sell (Current, issuer_id)
    end
end
```

An investor has features to buy and sell shares. To execute one of these features, the investor must wait for the lock on the market and for the precondition to be satisfied. In the feature call to the market, the investor passes a reference to itself, which triggers lock passing. This enables the market to query the investor's identifier in a separate callback, but the lock passing operation forces the investor to wait until the market finished. Consider the following routine, which makes an investor buy a share on a market. A log keeps a record of the transaction.

```
do_transaction (
  investor: separate INVESTOR;
  issuer_id: INTEGER;
  log: separate LOG
)
    −− Make the investor buy a share of the issuer on a
        market. Log the transaction.
  require
    not log.is_full
  do
    investor.buy (market, issuer_id)
    log.add_buy_entry (investor.id, issuer_id)
  end
```

Fig. 2 illustrates a possible execution with one timeline for each processor. Looking at these timelines, one can deduce the time metrics shown in Definition 1.

Definition 1 (Time metrics). The following time metrics exist:

- The *lifetime* of a processor: the time from when the processor gets created until when it is no longer needed, at which point it will be reclaimed by the runtime system.
- The *queue time* of a feature request: the time from the feature call until the feature application can begin. For non-separate feature calls the queue time is zero.
- The *synchronization time* of a feature request: the time from the creation of the locking request until when the locking request is approved. In other words, the synchronization time is the time it takes to obtain all the required locks and to satisfy the precondition.
- The *execution time* of a feature request: the time during which the feature actually gets executed. It is divided into the following parts:

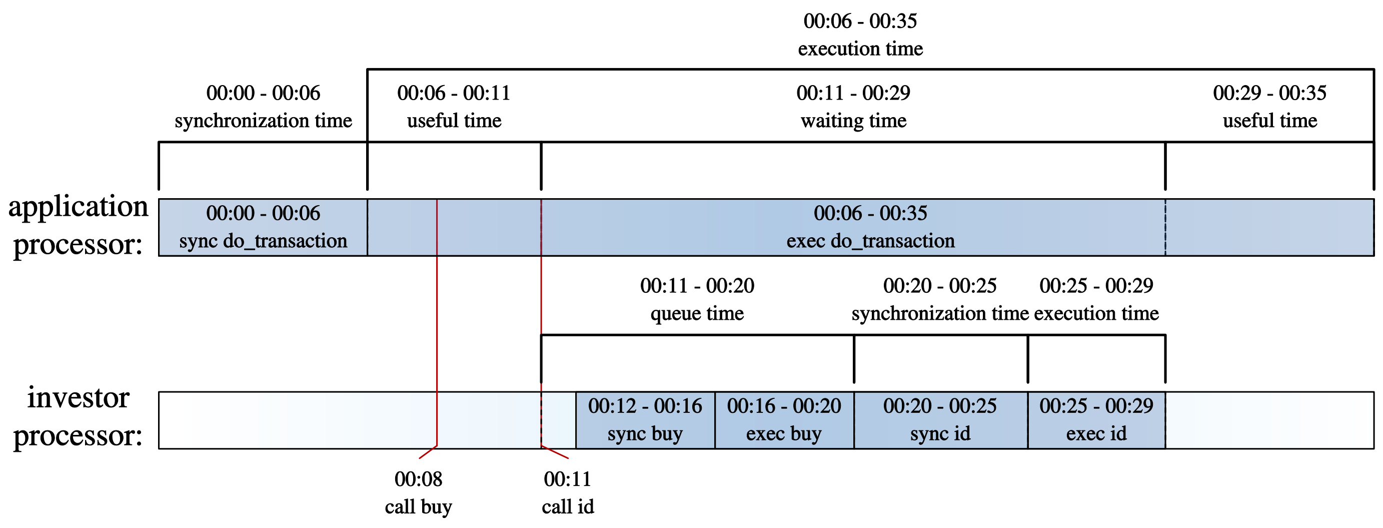


Fig. 2: Time intervals for the market example. The application first synchronizes (00:00 – 00:06) to get the locks on its arguments' handlers and to ensure that the log is not full. It then executes (00:06 – 00:35) two feature calls and waits for the second one to complete. Each feature call leads to a synchronization and execution step in the investor.

- The *waiting time* is the sum of:
    - the queue time, the synchronization time, and the execution time of each feature request that results from a synchronous separate feature call, and
    - the synchronization time of each feature request that results from a non-separate feature call.
- The *useful time*: the remaining part of the execution time in which the executing processor is not waiting.

- The *measuring time* of a processor: the time during which the processor performs measuring related actions.
- The *idle time* of a processor: the time during which the processor is not synchronizing, not executing, and not measuring.

During a synchronization step, the scheduler might evaluate the precondition multiple times until it finds it to hold. The scheduler used for this work processes locking requests in the order in which they arrived; when a processor unlocks, the scheduler starts from the beginning of the queue to check for earlier locking requests that now have become satisfiable. This algorithm causes the scheduler to evaluate the precondition of a locking request multiple times. To measure this, another metric counts the number of precondition evaluations in a synchronization step. Yet another metric counts the number of feature requests that have been generated for a feature; it is useful to calculate statistical values. These metrics are shown in Definition 2.

Definition 2 (Count metrics). The following count metrics exist:

- The *number of precondition evaluations* of a feature request: the number of times the scheduler evaluates the precondition in the synchronization step.
- The *number of feature requests* of a feature: the number of feature requests that have been generated for the feature.

The term *raw metrics* covers the time and count metrics introduced so far. With the number of feature requests one can construct the mean and the standard deviation of all metrics that are defined per feature request, i.e., queue time, synchronization time, execution time, waiting time, useful time, and the number of precondition evaluations. One can also sum up these metrics to obtain aggregated values. The term *derived metrics* covers the means, the standard deviations, and the aggregates of the raw metrics.

3.2. Using the metrics to diagnose issues and find solutions

The raw metrics are useful to diagnose issues in concurrent programs and find appropriate solutions. A large queue time of a feature request indicates that it takes a long time for the request to be served by the responsible processor and that therefore this processor is too busy. This can be resolved by distributing the workload onto more processors.

A large synchronization time of a feature request can have two reasons. If the feature request has a high number of precondition evaluations, then the scheduler is able to get the required locks, but it has problems to satisfy the precondition. If the feature request has a low number of precondition evaluations, then the required locks are hard to obtain because it is taking the scheduler a long time to even get to the point where it can evaluate the precondition. In the first case, one can look at the precondition to figure out why it is so hard to be satisfied. In the second case, one can introduce additional processors that replicate the roles of the congested processors.

A large execution time of a feature request is either due to a large waiting time or a large useful time. To reduce the waiting time, one can reduce the queue time, the synchronization

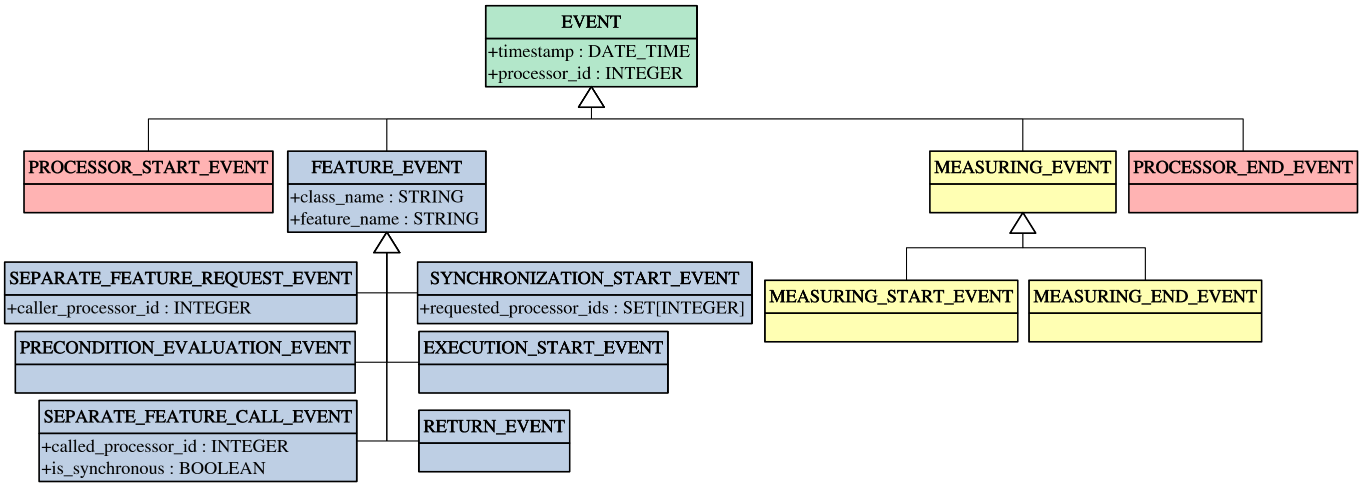

EVENT
+timestamp : DATE_TIME
+processor_id : INTEGER
PROCESSOR_START_EVENT
FEATURE_EVENT
+class_name : STRING
+feature_name : STRING
MEASURING_EVENT
PROCESSOR_END_EVENT
SEPARATE_FEATURE_REQUEST_EVENT
+caller_processor_id : INTEGER
SYNCHRONIZATION_START_EVENT
+requested_processor_ids : SET[INTEGER]
MEASURING_START_EVENT
MEASURING_END_EVENT
PRECONDITION_EVALUATION_EVENT
EXECUTION_START_EVENT
SEPARATE_FEATURE_CALL_EVENT
+called_processor_id : INTEGER
+is_synchronous : BOOLEAN
RETURN_EVENT

Fig. 3: Class hierarchy for events. The class *EVENT* is the parent of all event types. The class *FEATURE_EVENT* is the parent of all events that are related to metrics for feature requests and features; the class *MEASURING_EVENT* is the parent of all measuring event types.

time, or the execution time of the performed synchronous feature calls. Alternatively, one can also eliminate some of the synchronous feature calls altogether. For instance, one can increase the concurrency granularity by synchronizing less often. To reduce the useful time, one must optimize the feature itself.

A large measuring time can be influenced through parameters of the measuring process (see Section 4 and Section 5).

A large idle time means that a processor is not busy enough so that one can introduce more load for this processor. A large number of feature requests for a feature in combination with large queue times, synchronization times, or execution times indicates that it is especially worthwhile to optimize the feature.

3.3. From traces to metric values

This section shows how to calculate the raw metrics from a *trace*, i.e., a sequence of events corresponding to one execution of a program. Each type of event is described by a class, as shown in Fig. 3. An event has a timestamp and refers to one processor.

For a feature request r on processor p, one has to calculate the queue time, the synchronization time, the execution time, and the number of precondition evaluations; the execution time is divided into waiting time and useful time. The following events facilitate these metrics. If r is a separate feature request that p receives from another processor g, p records a *separate feature request event*. This event contains g's identifier. As soon as p starts the synchronization step for r, p creates a *synchronization start event*. Each such event contains the identifiers of the processors whose locks p requests. Each time the scheduler evaluates the precondition, it adds one *precondition evaluation event* on behalf of p. As soon as the locking request is approved, p starts with the execution. For this, p records an *execution start event*. If p makes a feature call to a different processor q, then processor p creates a *separate feature call event*. This event stores q's identifier and whether the feature call is synchronous or asynchronous. One can use q's identifier to find the events on q that are related to the resulting feature request by reconstructing each processor's request queue and call stack from the sequence of events. With this, one can link the separate feature call event on p to the related separate feature request event on q and then match this event to the remaining related events on q. After p is done with the execution, p creates a *return event*.

Fig. 4 shows the metrics calculation for r in more detail. If r is a separate feature request, then the queue time is the time between the separate feature request event and the synchronization start event; the queue time of a non-separate feature request is zero. The synchronization time is the time between the synchronization start event and the execution start event, and the execution time is the time between the execution start event and the return event. To calculate the waiting time of r, one first looks for all synchronous separate feature call events that p recorded during the execution step. The time between such a separate feature call event and the matching return event on the called processor is part of the waiting time. In addition, one looks for synchronization start events that p recorded during the execution step; these events belong to non-separate feature calls. The time between the synchronization start event and the matching execution start event is also part of the waiting time. The useful time is the part of the execution time that is not waiting time. The number of precondition evaluations is the count of precondition evaluation events.

For a feature f, one only has to calculate the number of feature requests for the feature. This number is simply the count of execution start events for f. Fig. 4 shows this calculation.

To calculate the lifetime of a processor p, p records an event when it gets created and when it is about to be disposed. Processor p's lifetime is the time difference between these two events. To calculate the measuring time, p creates an event when a measuring action starts and another event when the action ends. Finally, the idle time of p is its lifetime minus its measuring time minus the synchronization time and execution time of each of its feature requests. Fig. 4 shows this in more detail.

$$queue\_time(r) \overset{def}{=} \begin{cases} t_{synchronization\_start}(r) - t_{separate\_feature\_request}(r) & \textit{if } r \text{ is a separate feature request} \\ 0 & \textit{if } r \text{ is a non-separate feature request} \end{cases} \tag{1}$$

$$synchronization\_time(r) \overset{def}{=} t_{execution\_start}(r) - t_{synchronization\_start}(r) \tag{2}$$

$$execution\_time(r) \overset{def}{=} t_{return}(r) - t_{execution\_start}(r) \tag{3}$$

$$waiting\_time(r) \overset{def}{=} \sum_{i=1}^{n}(t_{return}(r_i) - t_{separate\_feature\_call}(r, w_i)) + \sum_{i=n+1}^{m}(t_{execution\_start}(r_i) - t_{synchronization\_start}(r_i))$$

where (4)

$w_1, \ldots, w_n$ are the synchronous separate feature calls issued while processing r

$r_1, \ldots, r_n$ are the matching feature requests on the called processors

$r_{n+1}, \ldots, r_m$ are the feature requests for non-separate feature calls issued while processing r

$$useful\_time(r) \overset{def}{=} execution\_time(r) - waiting\_time(r) \tag{5}$$

$$number\_of\_precondition\_evaluations(r) \overset{def}{=} c_{precondition\_evaluation}(r) \tag{6}$$

$$number\_of\_feature\_requests(f) \overset{def}{=} c_{execution\_start}(f) \tag{7}$$

$$lifetime(p) \overset{def}{=} t_{processor\_end}(p) - t_{processor\_start}(p) \tag{8}$$

$$measuring\_time(p) \overset{def}{=} \sum_{i=1}^{n}(t_{measuring\_end}(p, w_i) - t_{measuring\_start}(p, w_i))$$

where (9)

$w_1, \ldots, w_n$ are the measuring steps on p

$$\begin{aligned} idle\_time(p) &\overset{def}{=} lifetime(p) - measuring\_time(p) - \sum_{i=1}^{n}(synchronization\_time(r_i) + execution\_time(r_i)) \\ &= lifetime(p) - measuring\_time(p) - \sum_{i=1}^{n}(t_{return}(r_i) - t_{synchronization\_start}(r_i)) \end{aligned} \tag{10}$$

where

$r_1, \ldots, r_n$ are the feature requests on p

Fig. 4: Definitions of metric values for a processor p, a feature request r, and a feature f. The notation $t_n(a)$ denotes the timestamp of the event with name n and parameter a. The notation $c_n(a)$ denotes the count of events with name n and parameter a in the trace.

4. Tool overview

This section presents the tool that calculates and visualizes the metrics from Section 3.

The SCOOP compiler automatically generates a *statically instrumented* executable when it compiles a SCOOP project where the performance analyzer is activated. The compiler inserts *indirect instrumentation* code, augmenting the original source code with calls to a *performance analysis library*. With the help of the instrumentation code and the extended SCOOP runtime, the tool performs *exact monitoring*: it collects its information consistently at specified places. This approach is different from *statistical monitoring*, where a tool collects its information by periodically sampling the program state. Statistical monitoring is not suitable for the SCOOP performance analyzer. Most of the proposed metrics are defined over intervals. To compute an interval, it is essential to know when the interval starts and when it ends. With statistical monitoring, however, this information would not be available reliably because the tool could sample the program at the wrong points in time. For metrics that capture a single point in time, the issue does not occur. For instance, Cilk (Tallent and Mellor-Crummey, 2009) measures the number of idle CPU cores and the percentage of execution time used for management activities. Both of these metrics aggregate over single points in time and are therefore suitable for statistical monitoring. As an advantage, the overhead can be controlled by adjusting the sampling rate; this is not possible with exact monitoring.

The tool performs *tracing*: each processor keeps a log of activities in the form of an event sequence in an own buffer. The processor flushes the buffer to disk once the buffer is full. The buffer size can be changed in the project's configuration file. Small buffers lead to frequent flushes; they are useful to analyze non-terminating programs or to decrease the memory overhead at the expense of the time overhead. Large buffers lead to in-

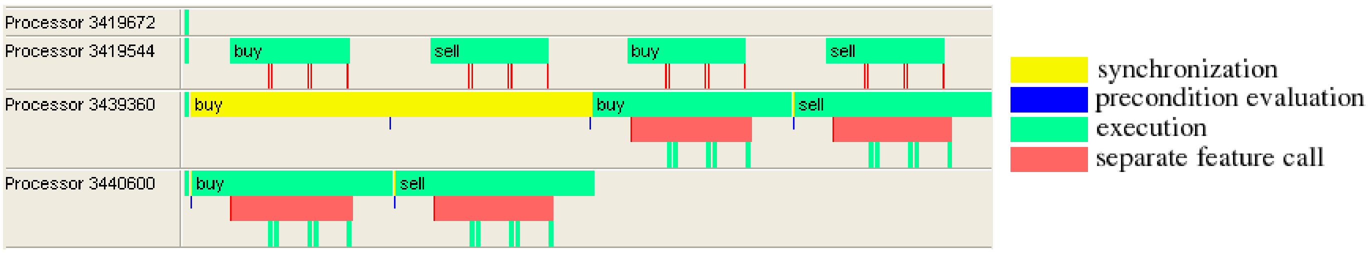


Fig. 5: Processor view for the share market example. The first timeline belongs to the root that created the market and the investors. The second timeline belongs to the market. The third and the fourth timelines belong to the two investors.

frequent flushes; they are useful to decrease the time overhead at the expense of the memory overhead. The resulting trace is not only useful to retroactively compute the metrics (see Section 3.3); it is also useful to visualize the execution and attribute the metric values to the different steps.

To reduce overhead, the tool does not analyze the trace *online*; instead it performs a *post-mortem analysis*. After an execution, the *analyzer* examines the trace. It first presents a list of all available traces; each trace is identified by the timestamp of its first event and its last event. For the chosen trace, the analyzer shows an interface to choose a time segment to be examined. Then it examines the chosen time segment and creates two views: the processor view and the feature view.

To explain the two views in detail, consider another transaction with one market, two investors, and one available share from one issuer. Each of the two investors wants first to buy the share and then to sell the share again.

```
do_transaction (
  first_investor: separate INVESTOR;
  second_investor: separate INVESTOR;
  issuer_id: INTEGER
)
    -- Make each of the two investors buy and then sell a
       share of the issuer on a market.
  do
    first_investor.buy (market, issuer_id)
    second_investor.buy (market, issuer_id)
    first_investor.sell (market, issuer_id)
    second_investor.sell (market, issuer_id)
  end
```

After the execution of this transaction each investor has two feature requests in its request queue: the first one for the *buy* feature and the second one for the *sell* feature.

4.1. Processor view

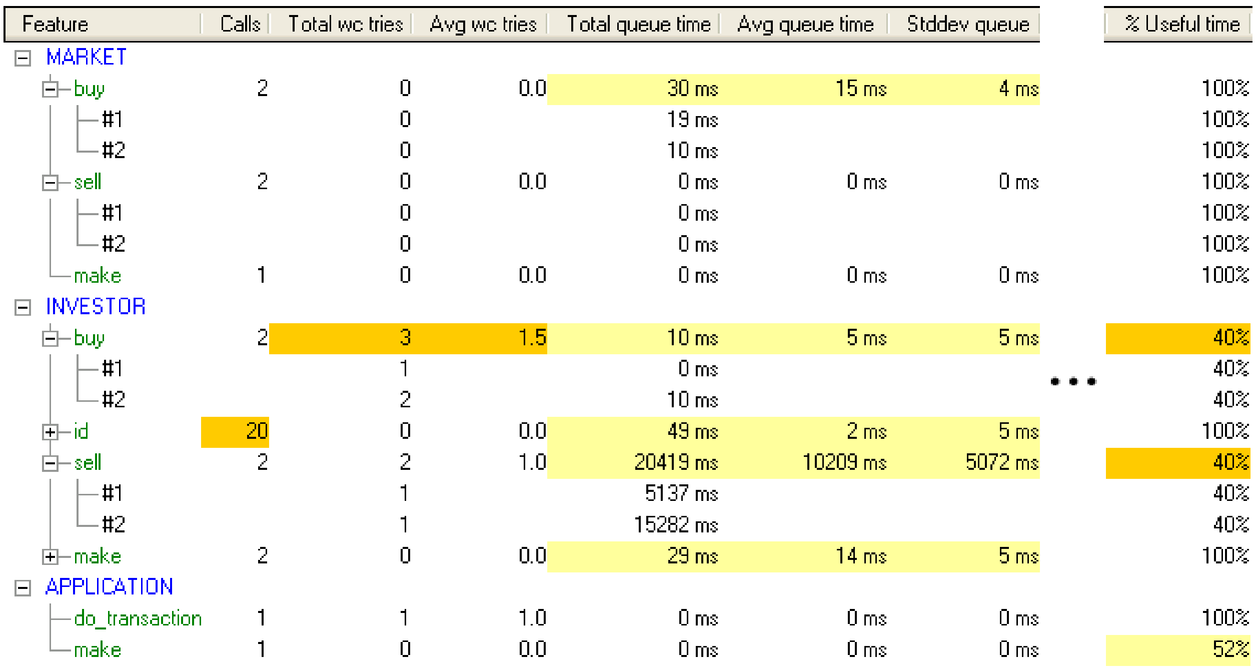
The *processor view* has one timeline for each processor; each of them shows all the actions performed by one processor. Fig. 5 shows the processor view for the market application.

Each step is presented as a bar, whose length indicates the duration of the action. The location of the bar in the timeline indicates when the action happened. The following bars exist:

- Each synchronization step is shown as a *synchronization bar*. This bar indicates the locked processors; it can be expanded to see the precondition evaluations as *precondition evaluation marks*.
- Each execution step is shown as an *execution bar*. This bar shows the metric values of the corresponding feature request. It can be expanded to see the feature calls: the tool shows the synchronization and execution step for each non-separate feature call and a *separate feature call bar* for each separate feature call. The separate feature call bar is linked to the resulting execution step; its length indicates the waiting time. Expanding the separate feature call bar shows the separate callbacks.
- Each measuring step is shown as a *measuring bar*.

Both investors sent a locking request for the execution of the *buy* feature to the scheduler. Both locking requests ask for the lock on the market and for the precondition of the *buy* feature to be satisfied. From the shorter synchronization bar, one can see that the scheduler approved the locking request of the second investor. The precondition evaluation mark below the synchronization bar indicates that the scheduler evaluated the precondition only once. During its execution step, the second investor performed a feature call to the *buy* feature on the market. This separate feature call was synchronous, as indicated by the elongated separate feature call bar below the execution bar. Below the separate feature call bar, the tool shows how the market made several separate callbacks to the *id* feature.

After the second investor finished the execution of the *buy* feature, it released the lock on the market and went straight into a second synchronization step for the *sell* feature. At this point, the scheduler had two locking requests: one for the *buy* feature of the first investor and one for the *sell* feature of the second investor. For fairness reasons, the scheduler first granted the lock on the market to the first investor and then evaluated the precondition. However, the scheduler was not able to satisfy the precondition; the precondition required that the first investor should have been able to buy a share. This was not possible at this point, because the share has been sold to the second investor. Hence, the scheduler took away the lock from the first investor and tried to satisfy the locking request of the second investor. For this, the scheduler granted the lock to the second investor and then evaluated the precondition. From the short synchronization step, one can tell that this was successful.



Feature	Calls	Total wc tries	Avg wc tries	Total queue time	Avg queue time	Stddev queue		% Useful time
MARKET								
buy	2	0	0.0	30 ms	15 ms	4 ms		100%
#1		0		19 ms				100%
#2		0		10 ms				100%
sell	2	0	0.0	0 ms	0 ms	0 ms		100%
#1		0		0 ms				100%
#2		0		0 ms				100%
make	1	0	0.0	0 ms	0 ms	0 ms		100%
INVESTOR								
buy	2	3	1.5	10 ms	5 ms	5 ms		40%
#1		1		0 ms			• • •	40%
#2		2		10 ms				40%
id	20	0	0.0	49 ms	2 ms	5 ms		100%
sell	2	2	1.0	20419 ms	10209 ms	5072 ms		40%
#1		1		5137 ms				40%
#2		1		15282 ms				40%
make	2	0	0.0	29 ms	14 ms	5 ms		100%
APPLICATION								
do_transaction	1	1	1.0	0 ms	0 ms	0 ms		100%
make	1	0	0.0	0 ms	0 ms	0 ms		52%

Fig. 6: Extract of the feature view for the market example. The measurements have been made on an Intel Core 2 Duo T9300 2.5 GHz CPU with 2 GB of RAM running Windows 7 SP1 (32 Bit). The system time resolution is 1 ms. Section 4.4 translates this resolution to the shown measurements.

After the execution of the *sell* feature, the second investor released the lock on the market processor. Finally, the scheduler was able to satisfy the locking request of the first investor. The processor view of the share market application does not list any measuring steps because we defined a large buffer size so that flushing does not happen in the shown time segment.

4.2. *Feature view*

The *feature view* groups feature requests according to the requested features. Fig. 6 show an extract of the feature view for the share market application.

The first column shows a number of trees; each tree corresponds to one class. Each first level node of a tree belongs to one feature of the class; such a *feature node* stands for a group of feature requests. Each second level node belongs to one of the feature requests; such a node is a *feature request node*. The remaining columns contain the metric values in both raw and derived form.

The tool highlights some of the metric values as hotspots. A *hotspot* is a metric value that might be a good starting point to optimize the program. There are two types of hotspots: worst performers and imperfections. A *worst performer* is the worst metric value in a metric column: the largest number of feature requests, the largest number of precondition evaluations, the biggest mean of precondition evaluations, and the smallest percentage of useful time. An *imperfection* is a metric value that is not ideal: a queue time, synchronization time, or execution time above zero, or a percentage of useful time below 100%. Imperfections are not as grave as worst performers; most reasonable features are imperfect. Thus, we suggest concentrating on the worst performers when optimizing.

4.3. *Using the tool to improve source code*

This section presents a guideline on how to use the tool to determine corrections in the source code. To start, one looks at the measuring bars in the processor view to determine whether the overhead is acceptable. If there are too many measuring bars, one must adapt the buffer size. Once the overhead is acceptable, one looks for hotspots in the feature view to know which feature to focus on. One then uses the processor view to find the executions of the troubled feature. Using the discussion from Section 3.2, one finds the issues and translates some of the proposed solutions to the source code. For this, one starts by opening the code of the troubled feature. The translation is not always straightforward; it requires developers that are knowledgeable about SCOOP. One might have to iterate over these steps several times; at the moment, there is no automatic solution.

In Fig. 6 one can see that both features *buy* and *sell* from class *INVESTOR* have on average 40% useful time. A low percentage of useful time is the result of waiting. For this, Section 3.2 suggests to look at the synchronous feature calls in *buy* and *sell*. From the processor view, one can tell that *buy* and *sell* perform synchronous feature calls to the market, causing separate callbacks to *id*. The feature view reflects this in the number of feature requests: the *id* feature has been called 20 times. This setup decreases the performance of the program in two ways. First, the feature call to the market is synchronous. Second, the separate callbacks cause unnecessary communication. To optimize the program, one can change the *buy* feature and the *sell* feature so that the investors pass the identifier instead of the investor object itself. This change turns the synchronous feature calls into asynchronous ones; the investors do not have to wait any longer and the market does not have to perform any separate callbacks.

4.4. Storing and loading events

Each processor keeps its events in its own buffer to avoid expensive synchronization actions on a central storage during the execution. The tool uses the timestamps to reconcile the events. For the non-distributed implementation of SCOOP, it was sufficient to use the system time for the timestamps. As a consequence, the resolution of the time measurements depends on the system where the tool runs. With a system time resolution of n ms, a measured point in time can have an error of $\pm n/2$ ms. Hence a value for a metric consisting of a single interval can have an error of $\pm n$ ms; a value for a metric consisting of m intervals, such as the waiting time metric, can have an error of $\pm(m \times n)$. On our system, the resolution is 1 ms.

For future distributed implementations, the timestamp could be based on logical clocks such as Lamport clocks (Lamport, 1978) or vector clocks (Mattern, 1989) for the ordering of events and synchronized clocks using the Network Time Protocol (Mills, 2010) for the approximation of durations. In practice, the Network Time Protocol could be precise enough to order the events from different nodes.

Each processor writes the events from its buffer to the disk, once the buffer is full. To write the events to disk, it serializes the buffer with all its events; the result is one file that contains the whole buffer. After the execution, one has a set of files from each processor.

The analyzer finds the timestamp of the first event and the last event in the trace so that developers can choose a time segment to be analyzed. To avoid that the analyzer must deserialize all the buffers to find these timestamps, each processor encodes in the file name the timestamps of the first and the last event. The analyzer then looks at the file names to find out when the execution starts and when it ends; it does not have to open any of the files. Once the time segment is chosen, the analyzer identifies all files that contain events in the chosen time segment. To find these files, it once again looks at the file names.

5. Evaluation

This section reports on the evaluation of the metrics and the tool. The evaluation includes an analysis of the tool's time overhead in Section 5.1 as well as an application of the metrics and the tool on a concurrent robotic control software in Section 5.2.

5.1. Time overhead analysis

A number of SCOOP solutions to well-known synchronization problems (Downey, 2005) served as subjects to measure the time overhead of the tool. We chose these programs because they represent various interactions that can take place in concurrent programs. Furthermore, these programs do not depend on external input, which makes them suitable for overhead measurements. However, we do not claim that these programs represent all possible interaction schemes. Fig. 7 shows the average time overhead for the programs over different buffer sizes. The average overhead is more interesting than the overhead of a single run because of the inherent nondeterminism in concurrent programs.

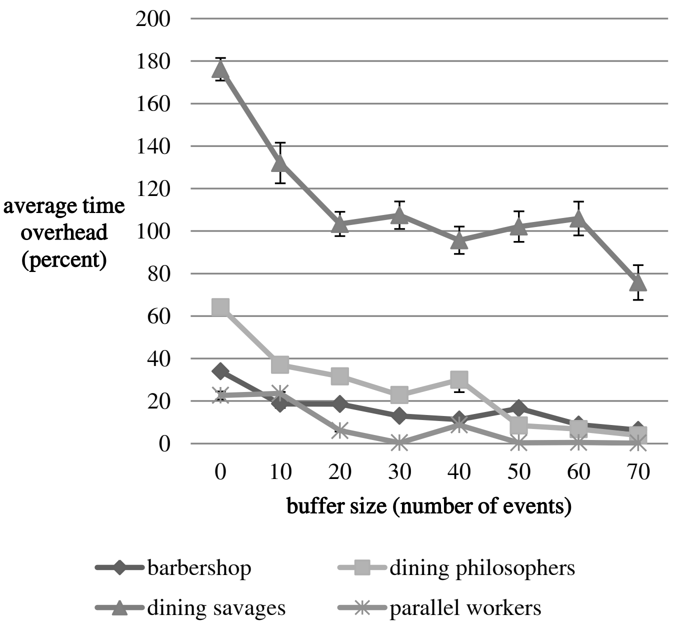


Fig. 7: Average time overhead for different programs and varying buffer sizes. Each point reflects the average over ten runs; the bars represent the standard deviations. The baseline is the average of ten runs where the performance analyzer is disabled. A buffer size of zero means that there is no buffer; each event gets immediately written to disk after creation. The experiments have been conducted on an Intel Core 2 Duo T9300 2.5 GHz CPU with 2 GB of RAM running Windows 7 SP1 (32 Bit).

Each of the programs produces a different time overhead. The reason comes from the different number of events per second: the more events a program produces per second, the higher the time overhead is. The dining savages program is the most intensive program because of the high number of exhibited interactions. The other programs show a more consistent behavior.

With growing buffer size, the time overhead of the programs decreases modulo a few bumps. The time overhead results from collecting the events in the buffers and from writing the buffers to disk; the second part contributes most to the time overhead. Since the buffer flush frequency decreases (non-strictly) with growing buffer size, the time overhead decreases (non-strictly) with growing buffer size as well. At some point the buffer size is large enough to hold all generated events, so that the tool can write all events at the end of the program; at this point the overhead is minimal. One can observe this saturation especially well with the parallel workers program. The bumps can be explained with the non-strictly decreasing flush frequency: a bump is the result of an increasing time overhead due to a larger buffer size that does not get compensated by a decreasing flush frequency.

While the decentralized event collection would support an increase of processors along with an increase of CPU cores, a program with thousands of processors is not representative because the current central scheduler of SCOOP does not support that many processors (see Section 2.2). Such a program would also require a different approach to visualization: it would not be reasonable for developers to analyze in detail the metric values for thousands of processors; instead, developers would require a high-level view on the metric values. On our system, we found that the SCOOP scheduler is able to handle up to one hundred processors, and we found that the tool remains usable.

In summary, one should always adapt the buffer size according to the programs characteristics in order to minimize the time overhead. For example, one can start with a small buffer size and use the processor view to judge whether there are too many measuring steps. One can then either increase or decrease the buffer size. The biggest improvement can be expected from an increase of the buffer size from zero to a bigger value.

5.2. Optimizing concurrent robotic control software

This section demonstrates how the metrics and the tool can be used to optimize a concurrent robotic control software. Robotic control software is especially suited for evaluation purposes, as the SCOOP model has its main strength in expressing concurrent interactions (handling the nondeterministic arrival of events), rather than purely deterministic parallel algorithms.

5.2.1. Hexapod

Earlier work (Ramanathan et al., 2010) demonstrates that SCOOP is suitable to implement concurrent robotic control software in a way that the code has a close correspondence to the behavioral specification. The hexapod robot, shown in Fig. 8, serves as an illustration for this. The hexapod has six

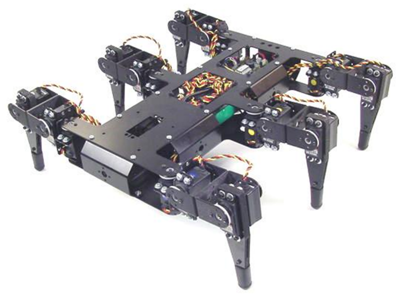
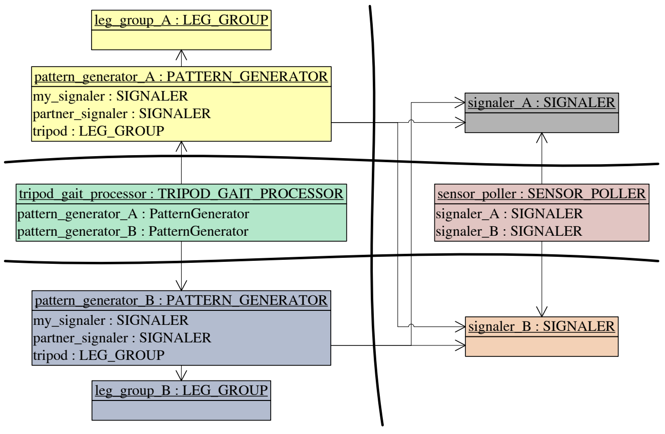


Fig. 8: The hexapod has six legs with three degrees of freedom.

legs. Each leg has three actuators, which represent the three degrees of freedom. The six legs are grouped into two tripods. Each tripod consists of the middle leg on one side and the outer legs on the other side; the legs in one tripod are synchronized with each other. To determine whether a tripod is planted on the ground, the tripod's middle leg is equipped with a force sensor; the tripod is on the ground whenever the load sensor reports a weight. In addition to the load sensor, the middle leg also has two angle sensors to detect when the tripod reached its extreme positions in the back or in the front. The hexapod has a simple interface to retrieve the sensor values and to send commands to the actuators. This interface is accessible over an integrated ZigBee wireless port.

The control software, shown in Fig. 9, runs on a PC with an external ZigBee port that connects to the hexapod. At runtime,


leg_group_A : LEG_GROUP
pattern_generator_A : PATTERN_GENERATOR
my_signaler : SIGNALER
partner_signaler : SIGNALER
tripod : LEG_GROUP
signaler_A : SIGNALER
tripod_gait_processor : TRIPOD_GAIT_PROCESSOR
pattern_generator_A : PatternGenerator
pattern_generator_B : PatternGenerator
sensor_poller : SENSOR_POLLER
signaler_A : SIGNALER
signaler_B : SIGNALER
pattern_generator_B : PATTERN_GENERATOR
my_signaler : SIGNALER
partner_signaler : SIGNALER
tripod : LEG_GROUP
signaler_B : SIGNALER
leg_group_B : LEG_GROUP
Fig. 9: The runtime structure of the hexapod controller. The thick separators represent processor boundaries.

the controller has a sensor poller that periodically connects to the hexapod to retrieve the latest sensor values. The sensor poller updates two signalers, one for each tripod, that translate the sensor values into meaningful queries. For example, two such queries use the load sensor values to return whether the tripod is on the ground or not.

Two pattern generators use the signalers to control the movements of the two tripods by sending commands to the hexapod. Each tripod executes an alternating sequence of protraction and retraction. To initialize, the first tripod lifts, swings forward, and drops. Then one normal iteration begins: the second tripod lifts and swings forward while the first tripod pushes the body forward using its foot as the pivot; the second tripod drops as soon as the first tripod is done. The first tripod's action is a *protraction*; the second tripod's action is a *retraction*. In the next iteration, the two tripods change the roles.

Each pattern generator is responsible to execute the alternating sequence for one of the tripods. Each pattern generator has one feature for each phase of an iteration. Each of these features has a precondition to ensure that the tripods are in a state to start the actions. For example, a retraction of a tripod can only start when the tripod is on the ground and the other tripod is raised; otherwise the hexapod will either trip or drag its legs. The following feature shows this in detail:

```
execute_retraction (
  my_signaler: separate SIGNALER;
  partner_signaler: separate SIGNALER
)
    -- Retract the tripod.
  require
    my_signaler.legs_down
    partner_signaler.legs_up
  do
    tripod.retract (my_signaler.retraction_time)
  end
```

5.2.2. Optimization

The synchronization times of the pattern generator features relate to the smoothness of the hexapod's gait: when the synchronization time of a feature increases, then the corresponding phase gets delayed, and the gait becomes more uneven. It is therefore important to keep the synchronization time as low as possible. There are two main influences on the synchronization time: the hexapod's mechanical and electronic constraints as well as the polling frequency. The hexapod's constraints cannot be influenced by the controller; however, the polling frequency can be changed.

When the polling frequency increases, then the controller knows about the hexapod's state sooner, which reduces the synchronization time. On the other hand, the polling frequency should be kept as low as possible to reduce the CPU time of the sensor poller and to reduce the network traffic between the controller and the hexapod. The optimal polling frequency is therefore a trade-off.

The synchronization time metric can be used to find the optimal polling frequency. For this purpose it is necessary to introduce a delay into the code of the sensor poller. One can then use the tool to measure the synchronization times of the pattern generator features and reduce the polling frequency until the synchronization times increase. Fig. 10 shows the tool's reported average total synchronization time and the average polling frequency for different delays in the sensor poller. The progression of the average polling frequency confirms that increasing the delay indeed reduces polling. The course of the average total synchronization time shows that synchronization becomes slower as the polling gets reduced. It is however interesting to see that the first 20 ms of delay do not have a big impact on synchronization. One can therefore reduce the polling frequency with a minimal impact on the hexapod's gait.

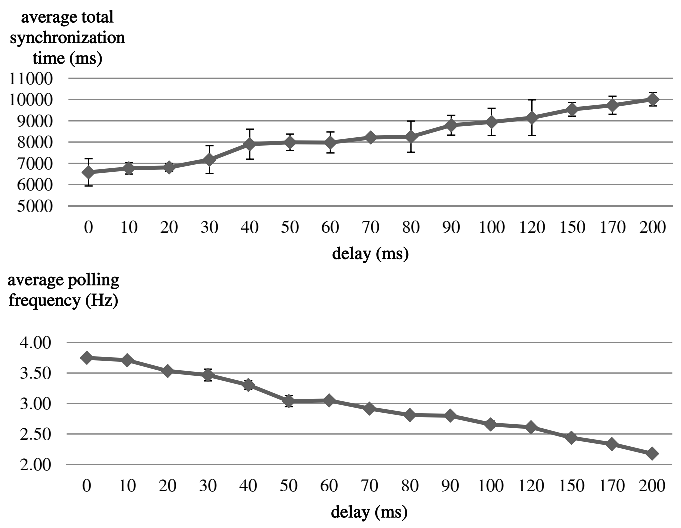


Fig. 10: Average total synchronization time and average polling frequency for varying polling delays. Each point reflects the average over three walks, each consisting of several iterations; the bars represent the standard deviations.

One concern remains to be answered: the tool is creating overhead, and thus it is not immediately clear whether the results are transferable to executions without the tool. In general, the overhead could cause the scheduler to switch to a fundamentally different schedule. In the case at hand, however, the scheduler has a limited choice. In the beginning, it can decide which tripod can start. Due to the symmetry of the hexapod (3 legs on each side), this decision does not change the essence of the schedule. Henceforth, the preconditions ensure that the scheduler lets the tripods proceed in the predefined order. For instance, the precondition of *execute_retraction* ensures that the retraction of one tripod only starts after the other tripod has lifted (see *partner_signaler.legs_up*). Similarly, another precondition ensures that one tripod can only drop after the other tripod has retracted. In fact, the only remaining scheduling ambiguity is whether one tripod starts swinging forward before the other tripod starts retracting, or the other way around. This choice, however, also happens without the tool and does not change the essence of the schedule.

The tool's overhead, however, causes an additional delay in the sensor poller. Due to this additional delay, the curves in Fig. 10 must be shifted to the right to transfer the results to executions without the tool. Furthermore, the overhead coming from the synchronization start events, the precondition evaluation events, and the execution start events affect the average total synchronization time. Hence, the curve with the average total synchronization time must be shifted to the bottom. Note that these two shifts are conservative in the sense that the above conclusions still hold on executions without the tool.

The optimization of the controller is not limited to the sensor poller. The number of feature requests also indicates that the controller performs a lot of unnecessary separate feature calls such as *my_signaler.retraction_time* in *execute_retraction*. As this feature call returns a value that is constant over an execution, the value can be cached in the pattern generators, which eliminates this and other similar calls. The tool is also helpful to visually validate the gait by inspecting the processor view for the controller.

6. Related work

This section compares the SCOOP performance analyzer and its metrics to other work. Tallent and Mellor-Crummey (2009) propose a performance analysis tool for Cilk (Blumofe et al., 1995) using statistical monitoring. The tool has two main metrics: *parallel idleness* of a program unit is the number of CPU cores that are idle during the execution; *parallel overhead* of a program unit is the percentage of execution time that is used for management activities. The metrics can be used to diagnose performance problems of a program unit. If the program unit has both low parallel idleness and low parallel overhead, then the parallelization is optimal. If only the parallel idleness is high, then one should refine concurrency granularity. If only the parallel overhead is high, then one should coarsen the concurrency granularity. If both metrics are high, then one should change the strategy; for instance, one can use a combination of data and task parallelism instead of just one of them. These metrics are well-suited for Cilk programs because concurrency is triggered implicitly by the runtime; thus it is indeed important to audit the runtime by measuring the CPU core utilization and the imposed overhead. These measurements are less important in SCOOP because developers trigger concurrency explicitly. This difference has another implication: the SCOOP performance analyzer breaks down the metric values along a timeline to show the explicitly triggered concurrent executions. In contrast, the Cilk performance analyzer shows the metric values along a call graph because developers do not need to know in detail how the runtime executes.

Su et al. (2010) present a performance analysis tool for programs based on the Partitioned Global Address Space (PGAS) model. A PGAS system has a global memory address space that is partitioned among nodes; all nodes can access all partitions. The proposed tool has an interface that PGAS systems implement to provide the necessary measurements. The tool offers metrics to measure the computational and storage load of nodes as well as the communication between different nodes. These metrics are well-suited for PGAS systems because they have a direct correspondence to the model. These metrics are also useful to describe the load and communication of SCOOP processors. However, they do not capture other aspects of the SCOOP model such as synchronization.

Wolf and Mohr (2003) introduce a performance analysis tool for MPI and OpenMP programs based on exact monitoring. The tool offers a number of metrics to measure communication inefficiencies, lock waiting time, barrier waiting time, and runtime overhead. For instance, the *late sender* metric represents the time wasted when a receiver executes a blocking receive operation before the sender starts. The metrics relate directly to concepts of the programming model and are thus helpful to find issues in programs. The communication inefficiencies metrics are similar to the queue time in the SCOOP performance analyzer because both represent the gap between a sender and a receiver; and the lock waiting time and the barrier waiting time metrics have a similar purpose than the synchronization time metric. These similarities are not surprising because most concurrent programming models have notions for communication and synchronization, and these notions must be reflected in any set of model-specific metrics.

Miller et al. (1995) present Paradyn – a performance analysis tool for MPI-based and multi-threaded programs. The tool uses dynamic instrumentation to perform statistical monitoring on demand; it is therefore suitable to analyze long-running programs. It supports a number of metrics to analyze communication and synchronization. The SCOOP performance analyzer uses static instrumentation due to lacking support in SCOOP; Paradyn relies on Dyninst (Hollingsworth et al., 1994) to dynamically instrument executables.

The Intel Parallel Amplifier (Intel, 2012) is a performance analysis tool for multi-threaded C++ and Fortran applications based on statistical monitoring. The tool calculates various architecture-specific metrics such as CPU core idleness and number of wrong branch predictions. It also has metrics that measure waiting of threads and locking of synchronization objects. The Intel tool is supported by a *Performance Monitoring Unit* that is part of Intel CPUs. The AMD CodeAnalyst (AMD, 2012) provides similar functionality for programs written in OpenCL, Java, .NET, C++, and Fortran running on AMD CPUs. Just as the Parallel Amplifier, the SCOOP performance analyzer keeps track of waiting on resources; it does so with the synchronization time and the number of precondition evaluations metrics. However, it does not record idle CPU cores and other architecture-specific values. Its metrics are all on the level of the programming model to make it easier to relate the metrics to the code.

Cai and Turner (1994) developed a monitor for programs written in *occam*, a concurrent programming language based on CSP (Hoare, 1985). The monitor minimizes the *probe effect* (the alteration of the system's behavior due to the measurement). It guarantees that in each execution the events are identical regardless of the monitor's presence. The monitor uses a variation of logical clocks (Lamport, 1978) to order events partially. Every process has one logical clock that records the execution time minus the delay caused by the monitor. Whenever a process wants to pick one channel from a set of available ones, the monitor delays the decision until it finds the communication partner that has a matching communication event with the lowest logical clock timestamp. To apply this approach to SCOOP, the scheduler would have to grant a locking request only when it is sure that it will not receive a locking request with an earlier timestamp. However, in SCOOP programs the set of processors that will issue locking requests is generally unknown. Hence, this approach is not directly applicable to SCOOP.

Hollingsworth and Miller (1992) present a quantitative technique to compare the effectiveness of concurrent performance metrics in guiding developers. The authors use this technique to compare the metrics of the following approaches:

- The *IPS-2* approach (Miller et al., 1990) computes for each method the useful CPU time.
- The *NPT* approach (Anderson and Lazowska, 1990) calculates for each method the time the executing process is doing useful work divided by the number of processes doing useful work at the same time. The approach suggests optimizing the method with the longest execution time and the largest number of processes that wait while the method is being executed.
- The critical path approach (Yang and Miller, 1988) determines the critical path and then sums up the useful CPU time for each method on the critical path.
- The *Logical Zeroing* approach (Miller et al., 1990) compares for each method the total execution time of the original critical path with the total execution time of the alternative critical path, where the duration of the method is set to zero. This gives an approximation of how much the total execution time will change when the method is optimized. The approach does not consider that the optimization might cause event reordering.
- The *Slack* approach (Hollingsworth and Miller, 1994) first determines the critical path. For each method on the critical path, the *slack value* indicates how much the method

can be improved before the critical path will change. The motivation is that an improvement of the longest executing method on the critical path might not significantly improve the total execution time because the critical path might change to a path that is slightly shorter. The approach does not work for applications where all paths are comparably long. It also ignores situations in which a method on the critical path is also on a secondary path and hence an improvement of the method will reduce the total execution time of both paths.

The authors conclude that there is no single universal metric. Performance analysis tools must support multiple metrics and assist in the selection. The metrics can be applied to most concurrency models. This makes them general, but also less expressive. For example, they are not designed to find synchronization issues, as done in the analysis of the hexapod (see Section 5). The SCOOP performance analyzer does not perform critical path analysis; however, it would be straightforward to calculate the critical path from the processor view.

The proposed metrics not only apply to SCOOP; they can also be adapted to concurrent programming models that share SCOOP's core ideas, i.e., assignment of an object to one autonomous thread of control along with a request queue and atomic locking with preconditions that have wait semantics. One such model is Cameo (Brooke and Paige, 2009), where each object has its own thread of control and an associated request queue for feature requests from other objects. All feature calls are synchronous unless the async keyword indicates otherwise. To perform a remote feature call, a client adds a feature request to the supplier's request queue. The behavior of a local feature call depends on the nature of the call: if the call is synchronous, the object processes the call immediately; if the call is asynchronous, the object adds a feature request to its own request queue. Before executing a feature, an object locks the actual arguments for the duration of the execution. It also delays the execution until the feature's wait condition is satisfied. When one of its suppliers is in need of its locks, it temporarily passes its locks to the supplier. The object implicitly obtains a *lazy lock* on a supplier that is not a locked actual argument; the lazy lock lasts just for the duration of the feature call on the supplier. A lazy lock is semantically equivalent to a synchronous feature call to a local wrapper with the same precondition as the called feature.

To adapt the proposed metrics to Cameo, one must translate from the SCOOP terminology to the Cameo terminology: separate feature calls become remote feature calls, non-separate feature calls become local feature calls, and processors become objects. In addition to these terminological changes, one must make a number of semantic changes. The *queue time* metric can be reused with a change: the queue time of a local feature call is not always zero; for asynchronous local feature calls, it is the time between the feature call and the beginning of the feature application. The *synchronization time* and *execution time* metrics can be used without a change; however, a lazy lock must be treated as an implicit local feature call with an own synchronization and execution time. Lastly, the *waiting time* metric must exclude asynchronous local feature calls because an object does not wait for itself to process such a call.

7. Conclusion

Different motivations exist for making a program concurrent: to optimize the performance, to increase the availability, to bring convenience to users, and to profit from the modeling power of concurrent programming languages. To assess concurrent programs with respect to the first goal, performance analysis tools are critical. Novel models, such as SCOOP, require new performance analysis tools because the execution semantics often differs considerably from established models. We presented a tool to find performance issues in SCOOP programs, using a number of novel metrics that are specific to the SCOOP model. For example, the synchronization time metric represents the time it takes to obtain the locks and to satisfy the precondition. In case of the robotic control example, this metric relates directly to the smoothness of the robot's gait. Using this example, we showed that the metrics are useful to optimize concurrent software. In future work, we want to introduce further metrics such as the number of processors along with the number of associated objects over time. We also want to introduce a time metric that measures the time it takes to evaluate a postcondition. For future distributed implementations of SCOOP, it will be necessary to also measure the impact of the network. For instance, latency could be shown as a percentage of the queue time, synchronization time, and waiting time.

Acknowledgments. We thank the anonymous reviewers for their insightful comments. We also thank Martino Trosi who has worked on the SCOOP performance analyzer as part of his Master's thesis. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, and ETH (ETHIIRA). Earlier work on the SCOOP model has also benefited from grants from the Swiss National Foundation and Microsoft (Multicore award).

References

AMD, 2012. CodeAnalyst. http://developer.amd.com/tools/CodeAnalyst/.

Anderson, T. E., Lazowska, E. D., 1990. Quartz: a tool for tuning parallel program performance. In: ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. pp. 115–125.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., Zhou, Y., 1995. Cilk: an efficient multithreaded runtime system. ACM SIGPLAN Notices 30 (8), 207–216.

Brooke, P. J., Paige, R. F., 2009. Cameo: an alternative model of concurrency for Eiffel. Formal Aspects of Computing 21, 363–391.

Brooke, P. J., Paige, R. F., Jacob, J. L., 2007. A CSP model of Eiffel's SCOOP. Formal Aspects of Computing 19 (4), 487–512.

Cai, W., Turner, S. J., 1994. An approach to the run-time monitoring of parallel programs. The Computer Journal 37 (4), 333–345.

Downey, A. B., 2005. The Little Book of Semaphores, 2nd Edition. Green Tea Press.

Eiffel Software, 2012. EiffelStudio. http://www.eiffel.com/.

ETH Zurich, 2012a. EVE. http://eve.origo.ethz.ch/.

ETH Zurich, 2012b. SCOOP. http://scoop.origo.ethz.ch/.

Hoare, C. A. R., 1985. Communicating Sequential Processes. Prentice Hall.

Hollingsworth, J. K., Miller, B. P., 1992. Parallel program performance metrics: A comparison and validation. In: ACM/IEEE Conference on Supercomputing. pp. 4–13.

Hollingsworth, J. K., Miller, B. P., 1994. Slack: a new performance metric for parallel programs. Tech. rep., University of Wisconsin Madison.

Hollingsworth, J. K., Miller, B. P., Cargille, J., 1994. Dynamic program instrumentation for scalable performance tools. In: Scalable High Performance Computing Conference. pp. 841–850.

Intel, 2012. Parallel Amplifier. http://software.intel.com/en-us/articles/intel-parallel-amplifier/.

Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21 (7), 558–565.

Mattern, F., 1989. Virtual time and global states of distributed systems. In: Workshop on Parallel and Distributed Algorithms. pp. 215–226.

Meyer, B., 1997. Object-Oriented Software Construction, 2nd Edition. Prentice-Hall.

Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., Newhall, T., 1995. The Paradyn parallel performance measurement tool. IEEE Computer 28, 37–46.

Miller, B. P., Clark, M., Hollingsworth, J. K., Kierstead, S., Lim, S.-S., Torzewski, T., 1990. IPS-2: the second generation of a parallel program measurement system. IEEE Transactions on Parallel and Distributed Systems 1 (2), 206–217.

Mills, D. L., 2010. Computer Network Time Synchronization: The Network Time Protocol on Earth and in Space, 2nd Edition. CRC Press.

Nanz, S., Nielson, F., Nielson, H. R., 2008. Modal abstractions of concurrent behaviour. In: International Static Analysis Symposium. pp. 159–173.

Nanz, S., Torshizi, F., Pedroni, M., Meyer, B., 2011. Design of an empirical study for comparing the usability of concurrent programming languages. In: International Symposium on Empirical Software Engineering and Measurement. pp. 325–334.

Nienaltowski, P., 2007. Practical framework for contract-based concurrent object-oriented programming. Ph.D. thesis, ETH Zurich.

Ostroff, J. S., Torshizi, F. A., Huang, H. F., Schoeller, B., 2008. Beyond contracts for concurrency. Formal Aspects of Computing 21 (4), 319–346.

Ramanathan, G., Morandi, B., West, S., Nanz, S., Meyer, B., 2010. Deriving concurrent control software from behavioral specifications. In: International Conference on Intelligent Robots and Systems. pp. 1994–1999.

Su, H.-H., Billingsley III, M., George, A. D., 2010. Parallel performance wizard: a performance system for the analysis of partitioned global-address-space applications. International Journal of High Performance Computing Applications 24, 485–510.

Tallent, N. R., Mellor-Crummey, J. M., 2009. Effective performance measurement and analysis of multithreaded applications. In: Principles and Practice of Parallel Programming. pp. 229–240.

West, S., Nanz, S., Meyer, B., 2010. A modular scheme for deadlock prevention in an object-oriented programming model. In: International Conference on Formal Engineering Methods. pp. 597–612.

Wolf, F., Mohr, B., 2003. Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture 49, 13–22.

Yang, C.-Q., Miller, B. P., 1988. Critical path analysis for the execution of parallel and distributed programs. In: International Conference on Distributed Computing Systems. pp. 366–373.

Benjamin Morandi received a Master in Computer Science from ETH Zurich in 2009. He is currently a Ph.D. student at the Chair of Software Engineering, ETH Zurich. His research interests include concurrent and distributed programming languages.

Sebastian Nanz is a postdoctoral researcher at the Chair of Software Engineering, ETH Zurich. His main research interests include concurrent systems and their verification, semantics of programming languages, and static analysis. Before joining ETH in 2009, he worked at the Technical University of Denmark and at Microsoft Research Cambridge. Sebastian holds a Ph.D. degree from Imperial College London and Master degrees in Mathematics and Computer Science from Technische Universität München.

Bertrand Meyer is Professor of Software Engineering at ETH Zurich, head of the Software Engineering Laboratory at ITMO (Saint Petersburg, Russia), and Chief Architect of Eiffel Software.